

A High-Level LTL Synthesis Format: TLSF v1.0

Swen Jacobs, Felix Klein

`{jacobs,klein}@react.uni-saarland.de`

Abstract

We present the Temporal Logic Synthesis Format (TLSF), a high-level format to describe synthesis problems via Linear Temporal Logic (LTL). The format builds upon standard LTL, but additionally allows to use high level constructs, such as sets and functions, to provide a compact and human readable representation. Furthermore, the format allows to identify parameters of a specification such that a single description can be used to define a family of problems. We also present a tool to automatically translate the format into plain LTL, which then can be used for synthesis by a solver. The tool also allows to adjust parameters of the specification and to apply standard transformations on the resulting formula.

1 Overview

We present the basic version of the Temporal Logic Synthesis Format (TLSF) in Sect. 2. In Sect. 3 we discuss the intended semantics of a specification, defined in terms of different implementation models. The full format is introduced in Sect. 4. The full format can be compiled automatically to the basic format. Thus, while it is more convenient to write specifications in the full format, for a synthesis tool it is sufficient to support the basic format. We illustrate the main features of the format in an example in Sect. 5. In Sect. 6, we give an overview of the SyFCo Tool, which can interpret the specification, possibly with respect to given parameter values, and transform it to the basic format, as well as a number of existing specification formats for backwards compatibility. Finally, we discuss possible extensions of the format in Sect. 7.

2 The Basic Format

A specification in the basic format consists of an INFO section and a MAIN section:

$$\langle info \rangle \langle main \rangle$$

2.1 The INFO Section

The INFO section contains the meta data of the specification, like a title and some description¹. Furthermore, it defines the underlying semantics of the specification (Mealy or Moore / standard or strict implication) and the target model of the synthesized implementation. Detailed information about supported semantics and targets can be found in Sect. 3. Finally, a comma separated list of tags can be specified to identify features of the specification, e.g., the restriction to a specific fragment of LTL. A $\langle tag \rangle$ can be any string literal and is not restricted to any predefined keywords.

¹We use colored verbatim font to identify the syntactic elements of the specification.

```

INFO {
  TITLE:      "<some title>"
  DESCRIPTION: "<some description>"
  SEMANTICS:   <semantics>
  TARGET:     <target>
  TAGS:       <tag>, <tag>, ...
}

```

2.2 The MAIN Section

The specification is completed by the MAIN section, which contains the partitioning of input and output signals, as well as the main specification, separated into environment assumptions, system invariants and system guarantees. Multiple declarations and expressions need to be separated by a ';'.

```

MAIN {
  INPUTS      { (<boolean signal declaration>)* }
  OUTPUTS     { (<boolean signal declaration>)* }
  ASSUMPTIONS { (<basic LTL expression>)* }
  INVARIANTS  { (<basic LTL expression>)* }
  GUARANTEES  { (<basic LTL expression>)* }
}

```

The ASSUMPTIONS, INVARIANTS and GUARANTEES subsections are optional.

2.3 Basic Expressions

A basic expression e is either a boolean signal or a basic LTL expression. Each basic expression has a corresponding type that is \mathbb{S} for boolean signals and \mathbb{T} for LTL expressions. Basic expressions can be composed to larger expressions using operators. An overview over the different types of expressions and operators is given below.

Boolean Signal Declarations. A signal identifier is represented by a string consisting of lowercase and uppercase letters ('a'-'z', 'A'-'Z'), numbers ('0'-'9'), underscores ('_'), primes (''), and at-signs ('@') and does not start with a number or a prime. Additionally, keywords like **X**, **G** or **U**, as defined in the rest of this document, are forbidden. An identifier is declared as either an input or an output signal. We denote the set of declared input signals as \mathcal{I} and the set of declared output signals as \mathcal{O} , where $\mathcal{I} \cap \mathcal{O} = \emptyset$. Then, a boolean signal declaration simply consists of a signal identifier $\langle name \rangle$ from $\mathcal{I} \cup \mathcal{O}$.

Basic LTL Expressions. A basic LTL expression conforms to the following grammar, including truth values, signals, boolean operators and temporal operators. For easy parsing of the basic format, we require fully parenthesized expressions, as expressed by the first of the following lines:

$$\begin{aligned}
\varphi &\equiv (\varphi') \\
\varphi' &\equiv \text{true} \mid \text{false} \mid s \quad \text{for } s \in \mathcal{I} \cup \mathcal{O} \mid \\
&\quad !\varphi \mid \varphi \&\& \varphi \mid \varphi \mid\mid \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \\
&\quad \text{X } \varphi \mid \text{G } \varphi \mid \text{F } \varphi \mid \varphi \text{ U } \varphi \mid \varphi \text{ R } \varphi \mid \varphi \text{ W } \varphi
\end{aligned}$$

Thus, a basic LTL expression is based on the expressions true, false, and signals, composed with boolean operators (negation, conjunction, disjunction, implication, equivalence) and temporal operators (next, globally, eventually, until, release, weak until). The semantics are defined in the usual way. A formal definition of the semantics of the temporal operators can be found in Appendix A.1.

3 Targets and Semantics

The TARGET of the specification defines the implementation model that a solution should adhere to. Currently supported targets are Mealy automata (**Mealy**), whose output depends on the current state and input, and Moore automata (**Moore**), whose output only depends on the current state. The differentiation is necessary since realizability of a specification depends on the target system model. For example, every specification that is realizable under Moore semantics is also realizable under Mealy semantics, but not vice versa. A formal description of both automata models can be found in Appendix A.2.

The SEMANTICS of the specification defines how the formula was intended to be evaluated, which also depends on the target implementation model. We currently support four different semantics: standard Mealy semantics (**Mealy**), standard Moore semantics (**Moore**), strict Mealy semantics (**Mealy,Strict**), and strict Moore semantics (**Moore,Strict**).

3.1 Basic cases

Standard semantics. If the semantics is (non-strict) **Mealy** or **Moore**, and the TARGET coincides with the semantics system model, then the specification is simply interpreted as an LTL formula. That is, if ASSUMPTIONS contains the LTL formula φ_e , INVARIANTS contains the formula ψ_s , and GUARANTEES contains the formula φ_s , then the specification is interpreted as

$$\varphi_e \rightarrow \mathbf{G} \psi_s \wedge \varphi_s$$

in standard LTL semantics (see Appendix A.1).

Strict semantics. If the semantics is **Mealy,Strict** or **Moore,Strict**, and the TARGET coincides with the semantics system model, then the specification is interpreted under strict implication semantics (as used in the synthesis of GR(1) specifications). Essentially, this means that the system is only allowed to violate safety properties if the environment has violated the safety assumptions at an earlier time. This is in contrast to the standard LTL semantics, where the specification is satisfied if the environment violates the assumptions at any time, regardless of the system behavior. For details, see Klein and Pnueli [7] and Bloem et al. [3].

3.2 Derived cases

Conversion between system models. If the implementation model of the SEMANTICS differs from the TARGET of a specification, we use a simple conversion to get a specification that is realizable in the target system model iff the original specification is realizable in the original system model: a specification in Moore semantics can be converted into Mealy semantics by prefixing all occurrences of input atomic propositions with an additional X-operator. Similarly, we can convert from Mealy semantics to Moore semantics by prefixing outputs with an X-operator.

Conversion from strict to non-strict. Finally, a specification in strict semantics can also be converted to an equivalent specification in standard semantics. Consider a specification with assumptions ρ_e , invariants ψ_s , and guarantees φ_s . Assume that ρ_e can be written as $\theta_e \wedge \mathbf{G} \psi_e \wedge \varphi_e$, where θ_e is an *initial constraint* (that only talks about the initial values of inputs), ψ_e is a *safety constraint* (we assume that it is an invariant over the current and next value of inputs at any time), and φ_e a *liveness constraint*. Similarly, assume that ψ_s is a safety constraint, and separate φ_s into an initial constraint θ_s and a liveness constraint φ'_s . Then the formula $\theta_e \wedge \mathbf{G} \psi_e \wedge \varphi_e \rightarrow \theta_s \wedge \mathbf{G} \psi_s \wedge \varphi_s$ under strict implication semantics can be converted to the following formula in standard semantics²:

$$\theta_e \rightarrow (\theta_s \wedge (\psi_s \mathcal{W} \neg \psi_e) \wedge (\mathbf{G} \psi_e \wedge \varphi_e \rightarrow \mathbf{G} \psi_s \wedge \varphi_s)).$$

²Note that in the conversion of [3], the formula is strengthened by adding the formula $\mathbf{G}(\mathbf{H} \psi_e \rightarrow \psi_s)$, where $\mathbf{H} \varphi$ is a Past-LTL formula and denotes that φ holds everywhere in the past. However, it is easy to show that our definition of strict semantics matches the definition of [3]. We prefer this notion, since it avoids the introduction of Past-LTL.

4 The Full Format

In the full format, a specification consists of three sections: the INFO section, the GLOBAL section and the MAIN section. The GLOBAL section is optional.

$$\langle info \rangle [\langle global \rangle] \langle main \rangle$$

The INFO section is the same as in the basic format, defined in Sect. 2.1. The GLOBAL section can be used to define parameters, and to bind identifiers to expressions that can be used later in the specification. The MAIN section is used as before, but can use extended sets of declarations and expressions.

We define the GLOBAL section in Sect. 4.1, and the changes to the MAIN section compared to the basic format in Sect. 4.2. The extended set of expressions that can be used in the full format is introduced in Sect. 4.3, extended signal and function declarations in Sections 4.4 and 4.5, and additional notation in Sections 4.6–4.8.

4.1 The GLOBAL Section

The GLOBAL section consists of the PARAMETERS subsection, defining the identifiers that parameterize the specification, and the DEFINITIONS subsection, that allows to define functions and bind identifiers to complex expressions. Multiple declarations need to be separated by a ';'. The section and its subsections are optional.

```
GLOBAL {
  PARAMETERS {
    (identifier = numerical expression);*
  }
  DEFINITIONS {
    ((function declaration) | (identifier = expression));*
  }
}
```

4.2 The MAIN Section

Like in the basic format, the MAIN section contains the partitioning of input and output signals, as well as the main specification. However, signal declarations can now contain signal buses, and LTL expressions can use parameters, functions, and identifiers defined in the GLOBAL section.

```
MAIN {
  INPUTS      { (signal declaration);* }
  OUTPUTS     { (signal declaration);* }
  ASSUMPTIONS { (LTL expression);* }
  INVARIANTS  { (LTL expression);* }
  GUARANTEES  { (LTL expression);* }
}
```

As before, the ASSUMPTIONS, INVARIANTS and GUARANTEES subsections are optional.

4.3 Expressions

An expression e is either a boolean signal, an n -ary signal (called bus), a numerical expression, a boolean expression, an LTL expression, or a set expression. Each expression has a corresponding type that is either one of the basic types: \mathbb{S} , \mathbb{U} , \mathbb{N} , \mathbb{B} , \mathbb{T} , or a recursively defined set type $\mathcal{S}_{\mathbb{X}}$ for some type \mathbb{X} .

As before, an identifier is represented by a string consisting of lowercase and uppercase letters ('a'-'z', 'A'-'Z'), numbers ('0'-'9'), underscores ('_'), primes (''), and at-signs ('@') and does not start with a number or a prime. In the full format, identifiers are bound to expressions of different type. We denote the respective sets of identifiers by $\Gamma_{\mathbb{S}}$, $\Gamma_{\mathbb{U}}$, $\Gamma_{\mathbb{N}}$, $\Gamma_{\mathbb{B}}$, $\Gamma_{\mathbb{T}}$, and $\Gamma_{\mathcal{S}_{\mathbb{X}}}$. Finally, basic expressions can be composed to larger expressions using operators. In the full format, we do not require fully parenthesized expressions. If an expression is not fully parenthesized, we use the precedence order given in Appendix A.3. An overview over the all types of expressions and operators is given below.

Numerical Expressions. A numerical expression $e_{\mathbb{N}}$ conforms to the following grammar:

$$\begin{aligned} e_{\mathbb{N}} \equiv & i \text{ for } i \in \Gamma_{\mathbb{N}} \mid n \text{ for } n \in \mathbb{N} \mid e_{\mathbb{N}} + e_{\mathbb{N}} \mid e_{\mathbb{N}} - e_{\mathbb{N}} \mid e_{\mathbb{N}} * e_{\mathbb{N}} \mid e_{\mathbb{N}} / e_{\mathbb{N}} \mid e_{\mathbb{N}} \% e_{\mathbb{N}} \\ & | e_{\mathcal{S}_{\mathbb{X}}} | \mid \text{MIN } e_{\mathcal{S}_{\mathbb{N}}} \mid \text{MAX } e_{\mathcal{S}_{\mathbb{N}}} \mid \text{SIZEOF } s \text{ for } s \in \Gamma_{\mathbb{U}} \end{aligned}$$

Thus, a numerical expression either represents an identifier (bound to a numerical value), a numerical constant, an addition, a subtraction, a multiplication, an integer division, a modulo operation, the size of a set, the minimal/maximal value of a set of naturals, or the size (i.e., width) of a bus, respectively. The semantics are defined in the usual way.

Set Expressions. A set expression $e_{\mathcal{S}_{\mathbb{X}}}$, containing elements of type \mathbb{X} , conforms to the following grammar:

$$\begin{aligned} e_{\mathcal{S}_{\mathbb{X}}} \equiv & i \text{ for } i \in \Gamma_{\mathcal{S}_{\mathbb{X}}} \mid \{e_{\mathbb{X}}, e_{\mathbb{X}}, \dots, e_{\mathbb{X}}\} \mid \{e_{\mathbb{N}}, e_{\mathbb{N}} \dots e_{\mathbb{N}}\} \mid \\ & e_{\mathcal{S}_{\mathbb{X}}} (+) e_{\mathcal{S}_{\mathbb{X}}} \mid e_{\mathcal{S}_{\mathbb{X}}} (*) e_{\mathcal{S}_{\mathbb{X}}} \mid e_{\mathcal{S}_{\mathbb{X}}} (\backslash) e_{\mathcal{S}_{\mathbb{X}}} \end{aligned}$$

Thus, the expression $e_{\mathcal{S}_{\mathbb{X}}}$ either represents an identifier (bound to a set of values of type \mathbb{X}), an explicit list of elements of type \mathbb{X} , a list of elements specified by a range (for $\mathbb{X} = \mathbb{N}$), a union of two sets, an intersection or a difference, respectively. The semantics of a range expression $\{x, y \dots z\}$ are defined for $x < y$ via:

$$\{n \in \mathbb{N} \mid x \leq n \leq z \wedge \exists j. n = x + j \cdot (y - x)\}.$$

The semantics of all other expressions is defined as usual. Sets contain either positive integers, boolean expressions, LTL expressions, buses, signals, or other sets of a specific type.

Boolean Expressions. A boolean expression $e_{\mathbb{B}}$ conforms to the following grammar:

$$\begin{aligned} e_{\mathbb{B}} \equiv & i \text{ for } i \in \Gamma_{\mathbb{B}} \mid e_{\mathbb{X}} \text{ IN } e_{\mathcal{S}_{\mathbb{X}}} \mid \text{true} \mid \text{false} \mid ! e_{\mathbb{B}} \mid \\ & e_{\mathbb{B}} \&\& e_{\mathbb{B}} \mid e_{\mathbb{B}} || e_{\mathbb{B}} \mid e_{\mathbb{B}} \rightarrow e_{\mathbb{B}} \mid e_{\mathbb{B}} \leftrightarrow e_{\mathbb{B}} \mid \\ & e_{\mathbb{N}} == e_{\mathbb{N}} \mid e_{\mathbb{N}} != e_{\mathbb{N}} \mid e_{\mathbb{N}} < e_{\mathbb{N}} \mid e_{\mathbb{N}} <= e_{\mathbb{N}} \mid e_{\mathbb{N}} > e_{\mathbb{N}} \mid e_{\mathbb{N}} >= e_{\mathbb{N}} \end{aligned}$$

Thus, a boolean expression either represents an identifier (bound to a boolean value), a membership test, true, false, a negation, a conjunction, a disjunction, an implication, an equivalence, or an equation between two positive integers (equality, inequality, less than, less or equal than, greater than, greater or equal than), respectively. The semantics are defined in the usual way. Note that signals are not allowed in a boolean expression, but only in an LTL expression.

LTL Expressions. An LTL expression φ conforms to the same grammar as a boolean expression, except that it additionally includes signals and temporal operators.

$$\varphi \equiv \dots \mid i \text{ for } i \in \Gamma_{\mathbb{T}} \mid s \text{ for } s \in \Gamma_{\mathbb{S}} \mid b[e_{\mathbb{N}}] \text{ for } b \in \Gamma_{\mathbb{U}} \mid \mathbf{X} \varphi \mid \mathbf{G} \varphi \mid \mathbf{F} \varphi \mid \varphi \mathbf{U} \varphi \mid \varphi \mathbf{R} \varphi \mid \varphi \mathbf{W} \varphi$$

Thus, an LTL expression additionally can represent an identifier bound to an LTL formula, a signal, an $e_{\mathbb{N}}$ -th signal of a bus, a next operation, a globally operation, an eventually operation, an until operation, a release operation, or a weak until operation, respectively. Note that every boolean expression is also an LTL expression, thus we allow the use of identifiers that are bound to boolean expressions as well. A formal definition of the semantics of the temporal operators can be found in Appendix A.1.

4.4 Signals and Buses

A signal declaration consists of the name of the signal. As for the basic format, signals are declared as either input or output signals, denoted by \mathcal{I} and \mathcal{O} , respectively. A bus declaration additionally specifies a signal width, i.e., a bus represents a finite set of signals.

$$\langle name \rangle \mid \langle name \rangle [e_{\mathbb{N}}]$$

In other words, a signal declaration \mathbf{s} specifies a signal $s \in \mathcal{I} \cup \mathcal{O}$, where a bus declaration $\mathbf{b}[n]$ specifies n signals $\mathbf{b}[0], \mathbf{b}[1], \dots, \mathbf{b}[n-1]$, with either $\mathbf{b}[i] \in \mathcal{I}$ for all i , or $\mathbf{b}[i] \in \mathcal{O}$ for all i .

Consider that we use $\mathbf{b}[i]$ to access the i -th value of b , i.e., we use the same syntax as for the declaration itself³. Also note that for the declared signals s , we have $s \in \mathcal{I} \cup \mathcal{O} \subseteq \Gamma_{\mathbb{S}}$, and for the declared buses b , we have $b \in \Gamma_{\mathbb{U}}$.

4.5 Function Declarations

As another feature, one can declare (recursive) functions of arbitrary arity inside the DEFINITIONS section. Functions can be used to define simple macros, but also to generate complex formulas from a given set of parameters. A declaration of a function of arity n has the form

$$\langle function\ name \rangle (\langle arg_1 \rangle, \langle arg_2 \rangle, \dots, \langle arg_n \rangle) = (e_c)^+,$$

where $\langle arg_1 \rangle, \langle arg_2 \rangle, \dots, \langle arg_n \rangle$ are fresh identifiers that can only be used inside the sub-expressions e_c . An expression e_c conforms to the following grammar:

$$e_c \equiv e \mid e_{\mathbb{B}} : e \mid e_{\mathbb{P}} : e \quad \text{where} \quad e \equiv e_{\mathbb{N}} \mid e_{\mathbb{B}} \mid e_{\mathbb{S}_{\mathbb{X}}} \mid \varphi$$

Thus, a function can be bound to any expression e , parameterized in its arguments, which additionally may be guarded by some boolean expression $e_{\mathbb{B}}$, or a pattern match $e_{\mathbb{P}}$. If the regular expression $(e_c)^+$ consists of more than one expression e_c , then the function binds to the first expression whose guard evaluates to **true** (in the order of their declaration). Furthermore, the special guard **otherwise** can be used, which evaluates to true if and only if all other guards evaluate to **false**. Expressions without a guard are implicitly guarded by **true**. All sub-expressions e_c need to have the same type \mathbb{X} . For every instantiation of a function by given parameters, we view the resulting expression $e_{\mathbb{X}}$ as an identifier in $\Gamma_{\mathbb{X}}$, bound to the result of the function application.

Pattern Matching. Pattern matches are special guards of the form

$$e_{\mathbb{P}} \equiv \varphi \sim \varphi',$$

which can be used to describe different behavior depending on the structure of an LTL expression. Hence, a guard $e_{\mathbb{P}}$ evaluates to **true** if and only if φ and φ' are structurally equivalent, with respect to their boolean and temporal connectives. Furthermore, identifier names that are used in φ' need to be fresh, since every identifier expression that appears in φ' is bound to the equivalent sub-expression in φ , which is only visible inside the right-hand-side of the guard. Furthermore, to improve readability, the special identifier `_` (wildcard) can be used, which always remains unbound. To clarify this feature, consider the following function declaration:

```
fun(f) =
  f ~ a U _ : a
  otherwise: X f
```

The function *fun* gets an LTL formula f as a parameter. If f is an until formula of the form $\varphi_1 \mathcal{U} \varphi_2$, then *fun*(f) binds to φ_1 , otherwise *fun*(f) binds to $X f$.

³C-Array Syntax Style

4.6 Big Operator Notation

It is often useful to express parameterized expressions using “big” operators, e.g., we use Σ to denote a sum over multiple sub-expressions, Π to denote a product, or \bigcup to denote a union. It is also possible to use this kind of notion in this specification format. The corresponding syntax looks as follows:

$$\langle op \rangle [\langle id_0 \rangle \text{ IN } e_{S_{x_0}}, \langle id_1 \rangle \text{ IN } e_{S_{x_1}}, \dots, \langle id_n \rangle \text{ IN } e_{S_{x_n}}] e_{\mathbb{X}}$$

Let x_j and S_j be the identifier represented by $\langle id_j \rangle$ and the set represented by $e_{S_{x_j}}$, respectively. Further, let \bigoplus be the mathematical operator corresponding to $\langle op \rangle$. Then, the above expression corresponds to the mathematical expression:

$$\bigoplus_{x_0 \in S_0} \bigoplus_{x_1 \in S_1} \dots \bigoplus_{x_n \in S_n} (e_{\mathbb{X}})$$

Note that $\langle id_0 \rangle$ is already bound in expression $e_{S_{x_1}}$, $\langle id_1 \rangle$ is bound in $e_{S_{x_2}}$, and so forth. The syntax is supported by every operator $\langle op \rangle \in \{+, *, (+), (*), \&\&, ||\}$.

4.7 Syntactic Sugar

To improve readability, there is additional syntactic sugar, which can be used beside the standard syntax. Let n and m be numerical expressions, then

- $\text{X}[n] \varphi$ denotes a stack of n next operations, e.g.:
 $\text{X}[3] \text{ a} \equiv \text{X X X a}$
- $\text{F}[n:m] \varphi$ denotes that φ holds somewhere between the next n and m steps, e.g.:
 $\text{F}[2:3] \text{ a} \equiv \text{X X(a || X a)}$
- $\text{G}[n:m] \varphi$ denotes that φ holds everywhere between the next n and m steps, e.g.:
 $\text{G}[1:3] \text{ a} \equiv \text{X(a \&\& X(a \&\& X a))}$
- $\langle op \rangle [\dots, n \circ_1 \langle id \rangle \circ_2 m, \dots] e_X$ denotes a big operator application, where $n \circ_1 \langle id \rangle \circ_2 m$ with $\circ_1, \circ_2 \in \{<, <=\}$ denotes that $\langle id \rangle$ ranges from n to m . Thereby, the inclusion of n and m depends on the choice of \circ_1 and \circ_2 , respectively. Thus, the notation provides an alternative to membership in combination with set ranges, e.g.:

$$\&\&[0 <= i < n] \text{ b}[i] \equiv \&\&[i \text{ IN } \{0, 1..n-1\}] \text{ b}[i]$$

4.8 Comments

It is possible to use C style comments anywhere in the specification, i.e., there are single line comments initialized by `//` and multi line comments between `/*` and `*/`. Multi line comments can be nested.

5 Example: A Parameterized Arbiter

To get some feeling for the interplay of the aforementioned features, consider the following example specification of a parameterized arbiter.

```

INFO {
  TITLE:      "A Parameterized Arbiter"
  DESCRIPTION: "An arbiter, parameterized in the number of clients"
  SEMANTICS:   Mealy
  TARGET:      Mealy
}

GLOBAL {
  PARAMETERS {
    // two clients
    n = 2;
  }
  DEFINITIONS {
    // mutual exclusion
    mutual(b) =
      || [i IN {0,1..n-1}]
        && [j IN {0,1..n-1} (\) {i}]
          !(b[i] && b[j]);
    // the Request-Response condition
    reqres(req,res) =
      G (req -> F res);
  }
}

/* Ensure mutual exclusion on the output bus and guarantee
   that each request on the input bus is eventually granted */
MAIN {
  INPUTS {
    r[n];
  }
  OUTPUTS {
    g[n];
  }
  INVARIANTS {
    mutual(g);
  }
  GUARANTEES {
    && [0 <= i < n]
      reqres(r[i],g[i]);
  }
}

```

The example is parameterized in the number of clients n (here: $n = 2$). Furthermore, it uses two functions: $\text{mutual}(b)$, which ensures mutual exclusions on the signals of a bus b of width n , and $\text{reqres}(req, res)$, which ensures that every request req is eventually followed by some response res . In the final specification, both conditions are then combined over the inputs r_i and outputs g_i .

6 The SyFCo Tool

We created a Synthesis Format Conversion Tool (SyFCo) [2] that can interpret the high level constructs of the format and supports transformation of the specification back to plain LTL. The tool has been designed to be modular with respect to the supported output formats and semantics. Furthermore, the tool can identify and manipulate parameters, targets and semantics of a specification on the fly, and thus allows comparative studies, as it is for example needed in the Synthesis Competition.

The main features of the tool can be summarized as follows:

- Evaluation of high level constructs in the full format to reduce full TLSF to basic TLSF.
- Transformation to other existing specification formats, like Promela LTL [1], PSL [5], Unbeast [4], or Wring [9].
- On the fly adjustment of parameters, semantics or targets.
- Preprocessing of the resulting LTL formula
 - conversion to negation normal form
 - replacement of derived operators
 - pushing/pulling next, eventually, or globally operators inwards/outwards
 - ...

7 Extensions

The format remains open for further extensions, which allow more fine grained control over the specification with respect to a particular synthesis problem. At the current time of writing, the following extensions were under consideration:

- Compositionality: The possibility to separate specifications into multiple components, which then can be used as building blocks to specify larger components.
- Partial Implementations: a specification that is separated into multiple components might also contain components that are already implemented. Implemented components could be given in the AIGER format that is already used in SYNTCOMP [6].
- Additional syntactic sugar, like enumerations or arithmetic on busses.

Acknowledgments

We thank Sebastian Schirmer, who supported us with the results of his Bachelor Thesis [8], to resolve many design decisions that came up during the development of this format. We thank Roderick Bloem, Rüdiger Ehlers, Bernd Finkbeiner, Ayrat Khalimov, Robert Könighofer, Nir Piterman, and Leander Tentrup for comments on the TLSF and drafts of this document.

References

- [1] Promela Manual Pages (Promela LTL). <http://spinroot.com/spin/Man/ltl.html>.
- [2] Synthesis Format Conversion Tool. <https://github.com/reactive-systems/syfco>.

- [3] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. *Synthesis of Reactive(1) designs*. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.
- [4] Rüdiger Ehlers. *Unbeast - Symbolic Bounded Synthesis*. <https://react.cs.uni-saarland.de/tools/unbeast>, 2010.
- [5] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer-Verlag, 2006.
- [6] Swen Jacobs. *Extended AIGER Format for Synthesis*. *CoRR*, abs/1405.5793, May 2014.
- [7] Uri Klein and Amir Pnueli. Revisiting synthesis of GR(1) specifications. In *HVC 2010. Revised Selected Papers*, volume 6504 of *LNCS*, pages 161–181. Springer, 2010.
- [8] Sebastian Schirmer. *A Specification Format for Reactive Synthesis*, March 2015. Bachelor Thesis.
- [9] Fabio Somenzi and Roderick Bloem. *Efficient Büchi Automata from LTL Formulae*. In *CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

A Appendix

A.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a temporal logic, defined over a finite set of atomic propositions AP. The syntax of LTL conforms to the following grammar:

$$\varphi ::= \text{true} \mid p \in \text{AP} \mid \neg\varphi \mid \varphi \vee \varphi \mid \text{X}\varphi \mid \varphi \mathcal{U} \varphi$$

The semantics of LTL are defined over infinite words $\alpha = \alpha_0\alpha_1\alpha_2 \cdots \in (2^{\text{AP}})^\omega$. A word α satisfies a formula φ at position $i \in \mathbb{N}$:

- $\alpha, i \models \text{true}$
- $\alpha, i \models p$ iff $p \in \alpha_i$
- $\alpha, i \models \neg\varphi$ iff $\alpha, i \not\models \varphi$
- $\alpha, i \models \varphi_1 \vee \varphi_2$ iff $\alpha, i \models \varphi_1$ or $\alpha, i \models \varphi_2$
- $\alpha, i \models \text{X}\varphi$ iff $\alpha, i+1 \models \varphi$
- $\alpha, i \models \varphi_1 \mathcal{U} \varphi_2$ iff $\exists n \geq i. \alpha, n \models \varphi_2$ and $\forall i \leq j < n. \alpha, j \models \varphi_1$

A word $\alpha \in 2^{\text{AP}}$ satisfies a formula φ iff $\alpha, 0 \models \varphi$. Beside the standard operators, we have the following derived operators:

- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$
- $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$
- $\text{F}\varphi \equiv \text{true} \mathcal{U} \varphi$
- $\text{G}\varphi \equiv \neg \text{F} \neg\varphi$
- $\varphi_1 \mathcal{R} \varphi_2 \equiv \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$
- $\varphi_1 \mathcal{W} \varphi_2 \equiv (\varphi_1 \mathcal{U} \varphi_2) \vee \text{G}\varphi_1$

A.2 Mealy and Moore Automata

A Mealy automaton is a tuple $\mathcal{M}_e = (\mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda_e)$, where

- \mathcal{I} is a finite set of input letters,
- \mathcal{O} is a finite set of output letters,
- Q is finite set of states,
- $q_0 \in Q$ is the initial state,
- $\delta: Q \times \mathcal{I} \rightarrow Q$ is the transition function, and
- $\lambda_e: Q \times \mathcal{I} \rightarrow \mathcal{O}$ is the output function.

Hence, the output depends on the current state of the automaton and the last input letter.

A Moore automaton is a tuple $\mathcal{M}_o = (\mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda_o)$, where $\mathcal{I}, \mathcal{O}, Q, q_0$ and δ are defined as for Mealy automata. However, the output function $\lambda_o: Q \rightarrow \mathcal{O}$ determines the current output only on the current state of the automaton, but not the last input.

A.3 Operator Precedence, Alternative Operators

The following table lists the precedence, arity and associativity of all expression operators. Also consider the alternative names in brackets which can be used instead of the symbolic representations.

| Precedence | Operator | Description | Arity | Associativity |
|------------|---|--|--------|---------------|
| 1 | <code>+[.]</code> (SUM[.]) <code>*[.]</code> (PROD[.]) <code> ... </code> (SIZE) <code>MIN</code> <code>MAX</code> <code>SIZEOF</code> | sum product size minimum maximum size of a bus | unary | |
| 2 | <code>*</code> (MUL) | multiplication | binary | left-to-right |
| 3 | <code>/</code> (DIV) <code>%</code> (MOD) | integer division modulo | binary | right-to-left |
| 4 | <code>+</code> (PLUS) <code>-</code> (MINUS) | addition difference | binary | left-to-right |
| 5 | <code>(*)[.]</code> (CAP[.]) <code>(+)[.]</code> (CUP[.]) | intersection union | unary | |
| 6 | <code>(\)</code> ((-),SETMINUS) | set difference | binary | right-to-left |
| 7 | <code>(*)</code> (CAP) | intersection | binary | left-to-right |
| 8 | <code>(+)</code> (CUP) | union | binary | left-to-right |
| 9 | <code>==</code> (EQ) <code>!=</code> (/=, NEQ) <code><</code> (LE) <code><=</code> (LEQ) <code>></code> (GE) <code>>=</code> (GEG) | equality inequality smaller than smaller or equal than greater then greater or equal than | binary | left-to-right |
| 10 | <code>IN</code> (ELEM, <-) | membership | binary | left-to-right |
| 11 | <code>!</code> (NOT) <code>X</code> <code>F</code> <code>G</code> <code>&&[.]</code> (AND[.], FORALL[.]) <code> [.]</code> (OR[.], EXISTS[.]) | negation next finally globally conjunction disjunction | unary | |
| 12 | <code>&&</code> (AND) | conjunction | binary | left-to-right |
| 13 | <code> </code> (OR) | disjunction | binary | left-to-right |
| 14 | <code>-></code> (IMPLIES) <code><-></code> (EQUIV) | implication equivalence | binary | right-to-left |
| 15 | <code>W</code> | weak until | binary | right-to-left |
| 16 | <code>U</code> | until | binary | right-to-left |
| 17 | <code>R</code> | release | binary | left-to-right |
| 18 | <code>~</code> | pattern match | binary | left-to-right |
| 19 | <code>:</code> | guard | binary | left-to-right |